PostgreSQL Partitioning and Sharding

Agenda

```
---------------------------------------------------------------------------
| 1. Introduction to PostgreSQL   | 10. Creating a Hash Partitioned Table |
| 2. What is Partitioning?        | 11. Defining Hash Partitions          |
| 3. Benefits of Partitioning     | 12. Creating a List Partitioned Table |
| 4. Types of Partitioning        | 13. Defining List Partitions          |
| 5. Partitioning Strategy        | 14. Range Partitioning Pros/Cons      |
| 6. Creating a Partitioned Table | 15. Hash Partitioning                 |
| 7. Defining Partitions          | 16. List Partitioning Pros/Cons       |
| 8. Inserting Data               | 17. Conclusion                        |
| 9. Querying Partitions          |                                       |
---------------------------------------------------------------------------
```

Slide 1: Introduction to PostgreSQL

- **PostgreSQL** is an open-source, object-relational database system.
- Advanced SQL compliance, reliability, feature robustness, and performance.
- Extensive support for complex queries, foreign keys, triggers, views, and stored procedures.

Slide 2: What is Partitioning?

- Division of a large database table into smaller, more manageable pieces.
- Each piece is called a "partition".
- Can be done by range, list, or hash method.

Slide 3: Benefits of Partitioning

- **Performance Improvement**: Enhances query performance through partition pruning.
- **Manageability**: Easier to manage and maintain smaller tables.
- **Data Retention**: Simplifies data archiving and purging processes.

Slide 4: Types of Partitioning

- **Range Partitioning**
  - Data is partitioned according to a range of values. Ideal for time-series data.
- **List Partitioning**
  - Data is partitioned based on a predefined list of values.
- **Hash Partitioning**
  - Data is partitioned based on a hash key.

Slide 5: Partitioning Strategy

1. **Choose Partition Key**
   - Commonly a date, ID, or status.
2. **Select Partition Type**
   - Range, List, or Hash.
3. **Define Partition Boundaries**
   - Based on the chosen key and type.

Slide 6: Creating a Partitioned Table

```
CREATE TABLE temperature_records (
    record_date DATE PRIMARY KEY,
    city_id INT,
    temperature DECIMAL
) PARTITION BY RANGE (record_date);
```

Slide 7: Defining Partitions

Partition for 2020

```sql
CREATE TABLE temperature_2020 PARTITION OF temperature_records
FOR VALUES FROM ('2020-01-01') TO ('2021-01-01');
```

Slide 8: Inserting Data

Inserting Data

- PostgreSQL automatically routes data to the correct partition.

```sql
INSERT INTO temperature_records VALUES (1, '2020-06-15', 75.0);
```

Slide 9: Querying Partitions

Querying Partitions

- Query the parent table as usual.

```sql
SELECT * FROM temperature_records WHERE record_date BETWEEN '2020-01-01' AND '2020-12-31';
```

Slide 10: Creating a Hash Partitioned Table

Partitioning a table by hash is useful for evenly distributing data across partitions, especially when there's no natural range or list to partition by.

```sql
CREATE TABLE user_sessions (
    session_id UUID,
    user_id INT,
    session_data JSONB,
    last_activity TIMESTAMP WITH TIME ZONE
) PARTITION BY HASH (user_id);
```

Slide 11: Defining Hash Partitions

```sql
CREATE TABLE user_sessions_1 PARTITION OF user_sessions FOR VALUES WITH (MODULUS 3, REMAINDER 0);
CREATE TABLE user_sessions_2 PARTITION OF user_sessions FOR VALUES WITH (MODULUS 3, REMAINDER 1);
CREATE TABLE user_sessions_3 PARTITION OF user_sessions FOR VALUES WITH (MODULUS 3, REMAINDER 2);
```

In this example, the `user_sessions` table is partitioned by hash based on the `user_id` column. The `MODULUS` value of 3 indicates that there are three partitions, and the `REMAINDER` value determines which partition a row belongs to based on the hash value of the `user_id`.

This approach ensures a more balanced distribution of data, which can be particularly beneficial for load balancing and improving query performance in scenarios where the distribution of data isn't naturally skewed towards certain values.

Slide 12: Creating a List Partitioned Table

Partitioning a table bylist is useful for having greater control over which records go to which partitions.

```
CREATE TABLE product_sales (
    product_id INT,
    store_id INT,
    sale_date DATE,
    quantity_sold INT
) PARTITION BY LIST (store_id);
```

Slide 13: Defining List Partitions

Create partitions for specific stores

```sql
CREATE TABLE product_sales_store_1 PARTITION OF product_sales FOR VALUES IN (1);
CREATE TABLE product_sales_store_2 PARTITION OF product_sales FOR VALUES IN (2);
CREATE TABLE product_sales_store_3 PARTITION OF product_sales FOR VALUES IN (3, 4, 5);
```

In this example, the `product_sales` table is partitioned by list, where specific values are assigned to each parition.

Pros:

- **Natural Ordering**: Particularly beneficial for time-series data where queries often target a specific range of dates, allowing for efficient data access and query performance.
- **Data Distribution Control**: Administrators can control data distribution based on the range, which helps in evenly distributing the data across partitions.
- **Efficient Data Purging**: Makes it easier to drop old data by simply dropping entire partitions, which can be more efficient than deleting rows from a large table.

Cons:

- **Uneven Data Distribution**: If the range values are not evenly distributed, some partitions may become significantly larger than others, leading to potential performance bottlenecks.

Pros:

- **Even Data Distribution**: Hash functions are designed to distribute data evenly across partitions, which can lead to more predictable performance across partitions.
- **Simplicity**: Requires less planning compared to range or list partitioning as the hash function automatically determines the partition.
- **Scalability**: Well-suited for large datasets where even distribution and scalability are critical.

Cons:

- **Less Intuitive for Range Queries**: Not ideal for queries that are range-based as the hash function scatters rows without regard to their logical order.
- **Hash Collisions**: Depending on the hash function and the number of partitions, hash collisions can occur, leading to potential imbalances in data distribution.

Pros:

- **Flexibility in Data Categorization**: Ideal for categorizing data into a known, finite list of categories, such as status codes or country names.
- **Simplifies Queries**: Can make queries simpler and more intuitive when filtering by categories that are directly mapped to partitions.
- **Data Isolation**: Useful for isolating and working with subsets of data, which can improve performance and manageability for specific query patterns.

Cons:

- **Limited Scalability**: As the list is predefined, adding a new category requires altering the table structure to add a new partition, which can be limiting and requires manual intervention.
- **Imbalanced Partitions**: Similar to range partitioning, if certain list categories have significantly more data than others, it can lead to imbalanced partitions

Slide 17: Conclusion

- **Partitioning** is a powerful strategy to manage large datasets in PostgreSQL.
- **Range**, **List**, and **Hash** partitioning methods offer different benefits and trade-offs.
- The choice of partitioning method depends on specific requirements like performance, scalability, and data distribution.
- Real-world scenarios might involve more complex partitioning schemes and data aggregation.
- **Sharding** is another strategy to distribute data across multiple database instances, which is particularly useful for high-traffic applications and distributed systems.

We will look at Sharding in the next section.